- **Introduction à Flutter :**
    - Présentation de Flutter et de ses fonctionnalités
    - Mise en place de l'environnement de développement
    - Introduction au langage de programmation Dart
- **Création d'interfaces utilisateur avec des widgets Flutter :**
    - Comprendre les widgets Flutter et sa hiérarchie
    - Création d'interfaces utilisateur de base à l'aide de widgets Flutter
    - Exploration de différents types de widgets (Container, Text, Image, etc.)
- **Mise en page et style dans Flutter :**
    - Comprendre le concept de mise en page Flutter
    - Utilisation de BoxConstraints et de widgets étendus
    - Application de styles à l'interface utilisateur à l'aide de thèmes et de feuilles de style
- **Entrée utilisateur et navigation :**
    - Gestion des entrées utilisateur à l'aide de widgets de formulaire
    - Naviguer entre les écrans à l'aide des itinéraires et du widget Navigator
    - Implémentation de Drawer, BottomNavigationBar et d'autres modèles de navigation
- **Utilisation des API et des données :**
    - Récupérer des données à partir d'API à l'aide de packages HTTP
    - Analyser les données JSON et les utiliser pour créer des interfaces utilisateur dynamiques

    - Stockage et récupération de données à l'aide du stockage local et des préférences partagées

- **Animations et Gestes :**

    - Implémentation d'animations de base à l'aide d'AnimatedContainer, AnimatedOpacity, etc.
    - Répondre aux gestes à l'aide des widgets GestureDetector et InkWell
    - Implémentation d'animations.

- **Sujets de Flutter avancés :**

    - Débogage et test des applications Flutter
    - Déploiement des applications Flutter dans les magasins d'applications
    - Construire des widgets personnalisés
    - Internationalisation et localisation

- **Projet**:

    - Construire une application Flutter du monde réel en utilisant les concepts appris dans le cours.

Présentation de Flutter et de ses fonctionnalités :

Flutter est un framework de développement mobile open-source et multiplateforme créé par Google. Il permet aux développeurs de créer des applications de haute qualité, similaires à des applications natives, pour les plateformes Android et iOS en utilisant un seul code.

Quelques-unes des principales fonctionnalités de Flutter incluent :

1. Développement rapide : Flutter propose un cycle de développement rapide avec "hot reload", qui permet aux développeurs de voir les modifications apportées à l'application en temps réel.
2. UI jolie et personnalisable : Flutter dispose d'une riche collection de widgets personnalisables qui donnent un aspect beau et cohérent à l'application.
3. Performances natives : Les applications Flutter sont compilées directement en code machine, ce qui garantit des performances rapides et fluides.
4. Programmation réactive : Flutter utilise la programmation réactive, ce qui permet de gérer de manière élégante et efficace les entrées utilisateur et les mises à jour de l'UI.
5. Accès aux fonctionnalités natives : Flutter a accès aux fonctionnalités et aux API natives, ce qui facilite la création d'applications qui nécessitent un accès aux fonctionnalités du dispositif telles que l'appareil photo, le microphone, etc.

Introduction au langage de programmation Dart

Dart est un langage de programmation open-source créé par Google. Il est utilisé pour développer des applications web et mobiles avec le framework Flutter. Dart a été conçu pour être facile à apprendre et à utiliser, avec une syntaxe familière pour les développeurs expérimentés.

Voici quelques-unes des caractéristiques principales de Dart :

1. Typage fort : Dart a un typage fort, ce qui signifie que les variables doivent être déclarées avec un type spécifique et ne peuvent pas être modifiées ultérieurement.
2. Classes et objets : Dart utilise la programmation orientée objet, avec des classes et des objets pour représenter les données et les fonctionnalités de l'application.
3. Fonctions fléchées : Dart prend en charge les fonctions fléchées (ou lambdas), qui permettent une syntaxe plus concise pour les opérations simples.
4. Compilation AOT et JIT : Dart peut être compilé en code machine soit à l'avance (AOT), soit à l'exécution (JIT), en fonction des besoins de performance de l'application.
5. Bibliothèques riches : Dart a une bibliothèque standard riche, avec des fonctionnalités telles que les collections, les expressions régulières, les sockets réseau, etc.

En utilisant Dart avec Flutter, les développeurs peuvent créer des applications mobiles performantes et attrayantes pour les plateformes Android et iOS.

Syntaxe de base et types de données dart :
Voici les éléments de base de la syntaxe Dart et les différents types de données disponibles :

1. Variables : Les variables peuvent être déclarées en utilisant le mot-clé "var", "final" ou "const". Le type de la variable est automatiquement inféré ou peut être explicitement spécifié. Par exemple :

```dart
var nom = "John";
final age = 30;
const pi = 3.14;
```

2. Types de données : Dart prend en charge les types de données suivants :

- Nombres (numériques) : int, double

- Chaînes de caractères : String
- Booléens : bool
- Lists (tableaux) : List
- Maps (dictionnaires) : Map
- Dynamic (dynamique) : permet de stocker n'importe quel type de données

Exemple :

```dart
int nombre = 42;
double pi = 3.14;
String nom = "John Doe";
bool estVrai = true;
List fruits = ['pomme', 'banane', 'cerise'];
Map utilisateur = {'nom': 'John', 'age': 30};
dynamic x = 42;
```

3. Opérateurs : Dart prend en charge les opérateurs usuels tels que +, -, *, /, etc. Il prend également en charge les opérateurs de comparaison tels que ==, !=, <, >, etc.
4. Conditionnelles : Les instructions conditionnelles telles que if et switch peuvent être utilisées pour contrôler le flux de l'application en fonction de certaines conditions.

```dart
if (age >= 18) {
  print("Vous êtes majeur.");
} else {
  print("Vous êtes mineur.");
}
```

5. Boucles : Les boucles telles que for, while et do-while peuvent être utilisées pour répéter des actions.

```dart
for (var i = 0; i < 5; i++) {
  print(i);
}
```

6. Fonctions : Les fonctions sont des blocs de code qui effectuent une tâche spécifique. Elles peuvent recevoir des arguments et retourner une valeur. Par exemple :

```dart
int carre(int nombre) {
  return nombre * nombre;
}

void afficherMessage(String message) {
  print(message);
}
```

Ce sont les éléments de base de la syntaxe Dart et des types de données. Il existe bien sûr de nombreuses autres fonctionnalités et concepts avancés, mais cela devrait vous donner une idée de ce à quoi vous pouvez vous attendre en travaillant avec Dart.

Classes, objets et héritage in french with examples :

1. Classes : Les classes sont des structures de données qui définissent les propriétés et les comportements d'un objet. Les classes peuvent contenir des variables d'instance, des méthodes et des constructeurs.

Exemple :

```
class Personne {
  String nom;
  int age;

  Personne(this.nom, this.age);

  void afficherInformations() {
    print("Nom : $nom");
    print("Age : $age");
  }
}
```

2. Objets : Les objets sont des instances de classes. Ils peuvent être créés en utilisant le mot-clé "new" suivi du nom de la classe.

Exemple :

```
var john = new Personne("John", 30);
john.afficherInformations();
```

3. Héritage : L'héritage est un mécanisme qui permet à une classe de hériter les propriétés et les comportements d'une autre classe. La classe qui hérite est appelée sous-classe et la classe à partir de laquelle l'héritage est effectué est appelée superclasse.

Exemple :

```
class Employe extends Personne {
  String poste;

  Employe(String nom, int age, this.poste) : super(nom, age);

  void afficherInformations() {
    super.afficherInformations();
    print("Poste : $poste");
  }
}

var jane = new Employe("Jane", 28, "Développeur");
jane.afficherInformations();
```

Les classes, les objets et l'héritage sont des concepts importants dans la programmation orientée objet. Ils peuvent vous aider à organiser et à structurer votre code de manière efficace et facile à gérer.

Comprendre les widgets Flutter et sa hiérarchie :

1. Widgets Flutter : Les widgets sont des éléments de base de l'interface utilisateur dans Flutter. Ils représentent des parties visuelles de l'application, telles que des boutons, des textes, des images, etc. Flutter propose une grande variété de widgets pour couvrir tous les besoins d'interface utilisateur.

Exemple :

```
Container(
  child: Text("Bonjour"),
  padding: EdgeInsets.all(16.0),
)
```

2. Hiérarchie de widgets : Tous les widgets Flutter sont organisés en une hiérarchie de widgets parent-enfant. Chaque widget est un enfant d'un autre widget, formant ainsi une arborescence de widgets. La racine de cette arborescence est généralement un widget "Scaffold" qui fournit une structure de base pour une application.

Exemple :

```
Scaffold(
  appBar: AppBar(
    title: Text("Mon application"),
  ),
  body: Container(
    child: Column(
      children: [
        Text("Section 1"),
        Text("Section 2"),
      ],
    ),
  ),
)
```

Comprendre la hiérarchie des widgets est important pour comprendre comment les widgets interagissent et comment organiser l'interface utilisateur de votre application. En utilisant les bons widgets dans la bonne hiérarchie, vous pouvez facilement créer des interfaces utilisateur intuitives et réactives.

Création d'interfaces utilisateur de base à l'aide de widgets

1. Widgets de base : Il existe de nombreux widgets de base que vous pouvez utiliser pour construire l'interface utilisateur de votre application, tels que : Text, Image, Container, Column, Row, etc. En utilisant ces widgets, vous pouvez construire une variété de mises en page et de structures d'interface utilisateur.

Exemple :

```
Container(
  child: Column(
    children: [
      Text("Titre"),
      Image.network("https://example.com/image.jpg"),
      Container(
        child: Text("Description"),
        padding: EdgeInsets.all(16.0),
      ),
    ],
  ),
  padding: EdgeInsets.all(16.0),
)
```

2. Gestion de l'état : Pour créer des interfaces utilisateur interactives, vous devez souvent gérer l'état de votre application. Cela signifie que vous devez pouvoir mettre à jour les widgets en fonction des actions de l'utilisateur. Flutter offre de nombreuses options pour gérer l'état de votre application, telles que le "StatefulWidget" et le "Bloc".

Exemple :

```dart
class MonCompteur extends StatefulWidget {
  @override
  _MonCompteurState createState() => _MonCompteurState();
}

class _MonCompteurState extends State<MonCompteur> {
  int _compteur = 0;

  void _augmenterCompteur() {
    setState(() {
      _compteur++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      child: Column(
        children: [
          Text("Compteur : $_compteur"),
          RaisedButton(
            child: Text("Augmenter"),
            onPressed: _augmenterCompteur,
          ),
        ],
      ),
      padding: EdgeInsets.all(16.0),
    );
  }
}
```

En utilisant les widgets de base et en gérant l'état de votre application, vous pouvez construire des interfaces utilisateur attrayantes et interactives dans Flutter.

Comprendre le concept de mise en page Flutter :

Le concept de mise en page en Flutter permet de déterminer comment les widgets sont disposés sur l'écran. Flutter offre une variété de widgets de mise en page pour aider à organiser les widgets, tels que :

1. Container : Permet d'encapsuler un widget en lui donnant une marge, un remplissage, une bordure et un fond.
2. Row et Column : Permettent de disposer des widgets les uns à côté des autres, soit en ligne (Row), soit en colonne (Column).
3. Expanded : Permet de faire en sorte que le widget s'étende pour remplir tout l'espace disponible dans la mise en page parente.
4. Stack : Permet de superposer des widgets les uns sur les autres.
5. ListView : Permet de disposer des widgets en liste défilante.

Exemple :

```
Container(
  child: Row(
    children: [
      Expanded(
        child: Container(
          color: Colors.red,
          height: 100.0,
        ),
      ),
      Expanded(
        child: Container(
          color: Colors.blue,
          height: 100.0,
        ),
      ),
    ],
  ),
)
```

En utilisant ces widgets de mise en page, vous pouvez construire une variété de mises en page pour votre application Flutter, en fonction de vos besoins.

Application de styles à l'interface utilisateur à l'aide de thèmes et de feuilles de style :

En Flutter, vous pouvez appliquer des styles à l'interface utilisateur en utilisant des thèmes et des feuilles de style.

Les thèmes sont des objets qui stockent les valeurs de couleur, de police et d'autres styles pour l'application. Ils peuvent être définis à l'aide de la classe ThemeData.

Exemple de thème :

```
ThemeData(
  primarySwatch: Colors.blue,
  fontFamily: 'Open Sans',
)
```

Les feuilles de style sont des fichiers qui peuvent être utilisés pour définir des styles pour des widgets spécifiques. Elles peuvent être créées en utilisant la classe StyleSheet.

Exemple de feuille de style :

```
const textStyle = TextStyle(
  fontSize: 20.0,
  fontWeight: FontWeight.bold,
  color: Colors.white,
);
```

Pour appliquer le style, vous pouvez utiliser la propriété "style" sur un widget Text.

Exemple :

```
Text(
  'Hello World',
  style: textStyle,
),
```

En utilisant les thèmes et les feuilles de style, vous pouvez facilement personnaliser l'apparence de votre application Flutter pour qu'elle corresponde à vos besoins.

Gestion des entrées utilisateur à l'aide de widgets de formulaire :

En Flutter, vous pouvez gérer les entrées utilisateur à l'aide de différents widgets de formulaire.

Voici quelques exemples :

1. TextField : Ce widget permet à l'utilisateur d'entrer du texte.

Exemple :

```
TextField(
  decoration: InputDecoration(
    labelText: 'Enter your name',
    border: OutlineInputBorder(),
  ),
)
```

2. Checkbox : Ce widget permet à l'utilisateur de sélectionner ou de désélectionner une option.

Exemple :

```
Checkbox(
  value: isChecked,
  onChanged: (value) {
    setState(() {
      isChecked = value;
    });
  },
)
```

3. Radio : Ce widget permet à l'utilisateur de sélectionner une option parmi plusieurs.

Exemple :

```
Radio(
  value: selectedRadio,
  groupValue: radioValue,
  onChanged: (value) {
    setState(() {
      radioValue = value;
    });
  },
)
```

4. DropdownButton : Ce widget permet à l'utilisateur de sélectionner une option à partir d'une liste déroulante.

Exemple :

```
DropdownButton(
  value: selectedOption,
  items: options.map((option) {
    return DropdownMenuItem(
      value: option,
      child: Text(option),
    );
  }).toList(),
  onChanged: (value) {
    setState(() {
      selectedOption = value;
    });
  },
)
```

En utilisant ces widgets de formulaire, vous pouvez facilement créer des formulaires interactifs dans votre application Flutter.

Naviguer entre les écrans à l'aide des itinéraires et du widget Navigator :
En Flutter, vous pouvez naviguer entre les écrans de votre application à l'aide du widget Navigator et des itinéraires.

1. Définition des itinéraires : Vous devez définir les itinéraires dans votre application en associant une chaîne de caractères unique à chaque écran.

Exemple :

```
final Map<String, WidgetBuilder> routes = {
  '/': (BuildContext context) => FirstScreen(),
  '/second': (BuildContext context) => SecondScreen(),
};
```

2. Utilisation du widget Navigator : Ensuite, vous pouvez utiliser le widget Navigator pour naviguer entre les écrans en utilisant les itinéraires que vous avez définis.

Exemple :

```
Navigator.pushNamed(context, '/second');
```

Vous pouvez également passer des données entre les écrans en utilisant les arguments du constructeur.

Exemple :

```
Navigator.pushNamed(
  context,
  '/second',
  arguments: SecondScreenArguments(data: 'Hello from first screen'),
);
```

En utilisant le widget Navigator et les itinéraires, vous pouvez facilement gérer la navigation dans votre application Flutter.

Implémentation de Drawer, BottomNavigationBar et d'autres modèles de navigation

Flutter propose plusieurs modèles de navigation pour organiser les écrans de votre application. Parmi les plus couramment utilisés, il y a Drawer, BottomNavigationBar et TabBar.

1. Drawer : Le Drawer est un menu latéral qui peut être déplié pour afficher les options de navigation supplémentaires dans votre application. Pour implémenter un Drawer, vous pouvez utiliser le widget Drawer et le fournir avec un Widget child.

Exemple :

```
Drawer(
  child: ListView(
    children: [
      ListTile(
        leading: Icon(Icons.home),
        title: Text('Home'),
        onTap: () {
          Navigator.pop(context);
          Navigator.pushNamed(context, '/');
        },
      ),
      ListTile(
        leading: Icon(Icons.settings),
        title: Text('Settings'),
        onTap: () {
          Navigator.pop(context);
          Navigator.pushNamed(context, '/settings');
        },
      ),
    ],
  ),
);
```

2. BottomNavigationBar : La BottomNavigationBar est une barre de navigation en bas de l'écran qui permet à l'utilisateur de naviguer entre les écrans en appuyant sur des icônes ou du texte. Pour implémenter une BottomNavigationBar, vous pouvez utiliser le widget BottomNavigationBar et fournir un tableau de BottomNavigationBarItem.

Exemple :

```
BottomNavigationBar(
  items: [
    BottomNavigationBarItem(
      icon: Icon(Icons.home),
      title: Text('Home'),
    ),
    BottomNavigationBarItem(
      icon: Icon(Icons.settings),
      title: Text('Settings'),
    ),
  ],
  currentIndex: _selectedIndex,
  onTap: (index) {
    setState(() {
      _selectedIndex = index;
    });
  },
);
```

En utilisant ces modèles de navigation, vous pouvez offrir une expérience de navigation cohérente et intuitive à vos utilisateurs dans votre application Flutter.

Récupérer des données à partir d'API à l'aide de packages HTTP

Avec Flutter, vous pouvez facilement accéder à des API externes et récupérer des données pour alimenter votre application. Pour cela, vous pouvez utiliser un package HTTP populaire tel que dart:io ou http. Voici un exemple de code qui montre comment effectuer une requête GET à une API et comment traiter les données reçues :

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;

Future<void> fetchData() async {
  final response = await http.get('https://api.example.com/data');

  if (response.statusCode == 200) {
    final data = json.decode(response.body);
    print(data);
  } else {
    throw Exception('Failed to load data');
  }
}
```

Dans ce code, nous utilisons http.get pour effectuer une requête GET à l'URL https://api.example.com/data. La méthode renvoie une réponse qui contient les données de l'API. Nous utilisons json.decode pour convertir les données de la réponse en un objet Dart, que nous pouvons ensuite utiliser pour alimenter notre application.

Voici un exemple plus complexe de comment récupérer des données à partir d'une API et les afficher dans une liste Flutter :

```dart
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

class APIExample extends StatefulWidget {
  @override
  _APIExampleState createState() => _APIExampleState();
}

class _APIExampleState extends State<APIExample> {
  var data;

  @override
  void initState() {
    super.initState();
    fetchData();
  }

  Future<void> fetchData() async {
    final response = await http.get('https://api.example.com/data');

    if (response.statusCode == 200) {
      setState(() {
        data = json.decode(response.body);
      });
    } else {
      throw Exception('Failed to load data');
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: data == null
          ? Center(child: CircularProgressIndicator())
          : ListView.builder(
              itemCount: data.length,
              itemBuilder: (context, index) {
                return ListTile(
                  title: Text(data[index]['title']),
                  subtitle: Text(data[index]['subtitle']),
                );
              },
            ),
    );
  }
}
```

Analyser les données JSON et les utiliser pour créer des interfaces utilisateur dynamiques

Pour analyser les données JSON et les utiliser pour créer des interfaces utilisateur dynamiques dans Flutter, vous pouvez suivre les étapes suivantes:

1. Définir le modèle de données : définir la structure des données en utilisant des classes Dart pour mieux gérer les données et les afficher dans l'interface utilisateur.
2. Charger les données à partir d'une API : utilisez les packages HTTP pour charger les données JSON à partir d'une API.
3. Analyser les données JSON : utilisez la bibliothèque dart convert pour analyser les données JSON et les convertir en objets Dart.
4. Afficher les données : utilisez les widgets appropriés pour afficher les données analysées sur l'interface utilisateur. Par exemple, vous pouvez utiliser des widgets ListView pour afficher des listes de données, ou des widgets Card pour afficher des informations détaillées sur un élément sélectionné.
5. Mettre à jour les données en temps réel : utilisez les techniques telles que le Binding de données pour mettre à jour automatiquement les données affichées sur l'interface utilisateur en fonction des données reçues.

Ce sont des étapes générales pour intégrer des données JSON à votre application Flutter. Les détails dépendent du modèle de données spécifique et de l'interface utilisateur que vous voulez créer.
Voici un exemple simple d'application Flutter qui utilise des données JSON pour afficher une liste d'articles :

1. Définition du modèle de données :

```dart
class Article {
  final String title;
  final String description;
  final String imageUrl;

  Article({this.title, this.description, this.imageUrl});

  factory Article.fromJson(Map<String, dynamic> json) {
    return Article(
      title: json['title'],
      description: json['description'],
      imageUrl: json['image_url'],
    );
  }
}
```

2. Chargement des données à partir d'une API :

```dart
Future<List<Article>> fetchArticles() async {
  final response = await http.get('https://your-api-url.com/articles');

  if (response.statusCode == 200) {
    List articlesJson = json.decode(response.body);
    return articlesJson.map((articleJson) =>
Article.fromJson(articleJson)).toList();
  } else {
    throw Exception('Failed to load articles');
  }
}
```

3. chage des données :

```dart
class ArticleListPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Articles'),
      ),
      body: FutureBuilder<List<Article>>(
        future: fetchArticles(),
        builder: (context, snapshot) {
          if (snapshot.hasData) {
            return ListView(
              children: snapshot.data.map((article) =>
ArticleCard(article)).toList(),
            );
          } else if (snapshot.hasError) {
            return Text("${snapshot.error}");
          }

          return CircularProgressIndicator();
        },
      ),
    );
  }
}

class ArticleCard extends StatelessWidget {
  final Article article;

  ArticleCard(this.article);

  @override
  Widget build(BuildContext context) {
    return Card(
      child: Column(
        children: [
          Image.network(article.imageUrl),
          Text(article.title),
          Text(article.description),
        ],
      ),
    );
  }
}
```

Cet exemple montre comment charger des données JSON à partir d'une API, les analyser en utilisant la bibliothèque convert et les afficher sur l'interface utilisateur à

l'aide de widgets Flutter. Vous pouvez adapter ce code en fonction de vos besoins pour créer des applications plus complexes.

Stockage et récupération de données à l'aide du stockage local et des préférences partagées

Pour stocker et récupérer des données dans Flutter, vous pouvez utiliser plusieurs options, comme le stockage local et les préférences partagées.

Pour le stockage local, vous pouvez utiliser la bibliothèque sqflite qui vous permet de stocker et récupérer des données dans une base de données SQLite. Voici un exemple de code pour stocker une liste de tâches dans une base de données SQLite:

```dart
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

class Task {
  int id;
  String name;

  Task({this.id, this.name});

  Map<String, dynamic> toMap() {
    return {
      'id': id,
      'name': name,
    };
  }
}

class TaskDatabase {
  static final TaskDatabase _instance = TaskDatabase._internal();
  factory TaskDatabase() => _instance;
  TaskDatabase._internal();

  static Database _db;

  Future<Database> get db async {
    if (_db != null) {
      return _db;
    }
    _db = await initDb();
    return _db;
  }

  initDb() async {
    String databasesPath = await getDatabasesPath();
    String path = join(databasesPath, 'tasks.db');

    var db = await openDatabase(path, version: 1, onCreate: _onCreate);
    return db;
  }

  void _onCreate(Database db, int version) async {
    await db.execute(
        "CREATE TABLE task (id INTEGER PRIMARY KEY, name TEXT)");
  }

  Future<Task> saveTask(Task task) async {
    var dbClient = await db;
    task.id = await dbClient.insert('task', task.toMap());
    return task;
  }

  Future<List<Task>> getTasks() async {
    var dbClient = await db;
    List<Map> maps = await dbClient.query('task', columns: ['id',
'name']);

    return maps.map((map) => Task.fromMap(map)).toList();
  }
}
```

Pour les préférences partagées, vous pouvez utiliser la bibliothèque shared_preferences. Voici un exemple de code pour stocker et récupérer une préférence simple:

```dart
import 'package:shared_preferences/shared_preferences.dart';

class Prefs {
  static final Prefs _instance = Prefs._internal();
  factory Prefs() => _instance;
  Prefs._internal();

  Future<bool> setString(String key, String value) async {
    final prefs = await SharedPreferences.getInstance();
    return prefs.setString(key, value);
  }

  Future<String> getString(String key) async {
    final prefs = await SharedPreferences.getInstance();
    return prefs.getString(key)
```

Implémentation d'animations de base à l'aide d'AnimatedContainer, AnimatedOpacity :

Les animations ajoutent du mouvement et de la vie à l'interface utilisateur et améliorent l'expérience utilisateur. Avec Flutter, les animations peuvent être créées de manière simple et rapide en utilisant les widgets mentionnés ci-dessus. Par exemple, on peut utiliser AnimatedContainer pour animer les changements de dimensions d'un widget, ou AnimatedOpacity pour animer les changements de transparence. Les exemples pourraient inclure l'animation d'un bouton sur un appui, ou la transition en douceur entre différents écrans.

Voici un exemple simple d'animation avec AnimatedContainer:

```dart
class MyAnimatedContainer extends StatefulWidget {
  @override
  _MyAnimatedContainerState createState() => _MyAnimatedContainerState();
}

class _MyAnimatedContainerState extends State<MyAnimatedContainer> {
  double _width = 50;
  double _height = 50;
  Color _color = Colors.green;

  void _changeDimensions() {
    setState(() {
      _width = 100;
      _height = 100;
      _color = Colors.red;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      width: double.infinity,
      height: 200,
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          AnimatedContainer(
            duration: Duration(seconds: 1),
            curve: Curves.fastOutSlowIn,
            width: _width,
            height: _height,
            color: _color,
          ),
          RaisedButton(
            child: Text("Animate"),
            onPressed: _changeDimensions,
          ),
        ],
      ),
    );
  }
}
```

Lorsque le bouton "Animate" est appuyé, les dimensions de la AnimatedContainer et sa couleur sont modifiées avec une animation d'une durée de 1 seconde et une courbe "fastOutSlowIn".

De la même manière, AnimatedOpacity peut être utilisé pour animer les changements de transparence:

```dart
class MyAnimatedOpacity extends StatefulWidget {
  @override
  _MyAnimatedOpacityState createState() => _MyAnimatedOpacityState();
}

class _MyAnimatedOpacityState extends State<MyAnimatedOpacity> {
  double _opacity = 1.0;

  void _changeOpacity() {
    setState(() {
      _opacity = 0.0;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      width: double.infinity,
      height: 200,
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          AnimatedOpacity(
            duration: Duration(seconds: 1),
            opacity: _opacity,
            child: Container(
              width: 100,
              height: 100,
              color: Colors.green,
            ),
          ),
          RaisedButton(
            child: Text("Animate"),
            onPressed: _changeOpacity,
          ),
        ],
      ),
    );
  }
}
```

Lorsque le bouton "Animate" est appuyé, la transparence de la Container est modifiée avec une animation d'une durée de 1 seconde.

Répondre aux gestes à l'aide des widgets GestureDetector et InkWell

Dans Flutter, les gestes sont gérés à l'aide de widgets tels que GestureDetector et InkWell. Le widget GestureDetector peut être utilisé pour détecter les gestes tels que le toucher, le double toucher, le glisser, etc. Par exemple, vous pouvez utiliser GestureDetector pour détecter un toucher sur un élément de l'interface utilisateur et effectuer une action en conséquence. Le widget InkWell fonctionne de manière similaire, mais il inclut également une animation de rebond pour donner une sensation de pression lors du toucher d'un élément.

Voici un exemple d'utilisation de GestureDetector :

```
GestureDetector(
  onTap: () {
    // Action à effectuer lorsque l'utilisateur touche l'élément
  },
  child: Container(
    width: 200.0,
    height: 100.0,
    color: Colors.blue,
    child: Text("Tap me!"),
  ),
)
```

Et voici un exemple d'utilisation de InkWell :

```
InkWell(
  onTap: () {
    // Action à effectuer lorsque l'utilisateur touche l'élément
  },
  child: Container(
    width: 200.0,
    height: 100.0,
    color: Colors.blue,
    child: Text("Tap me!"),
  ),
)
```

Implémentation d'animations

Pour implémenter des animations et des gestes personnalisés avec Flutter, vous pouvez utiliser les classes AnimationController, Tween et Animation. Vous pouvez également utiliser les widgets GestureDetector et InkWell pour gérer les entrées utilisateur.

Exemple d'animation personnalisée :

```dart
import 'package:flutter/material.dart';

class MyCustomAnimation extends StatefulWidget {
  @override
  _MyCustomAnimationState createState() => _MyCustomAnimationState();
}

class _MyCustomAnimationState extends State<MyCustomAnimation>
    with TickerProviderStateMixin {
  AnimationController _controller;
  Animation _animation;

  @override
  void initState() {
    _controller = AnimationController(
      vsync: this,
      duration: Duration(seconds: 2),
    );
    _animation = Tween(begin: 0.0, end: 1.0).animate(_controller);
    _controller.forward();
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return AnimatedBuilder(
      animation: _animation,
      builder: (context, child) {
        return Container(
          width: 200.0 * _animation.value,
          height: 200.0 * _animation.value,
          color: Colors.red,
        );
      },
    );
  }
}
```

Construire des widgets personnalisés

Pour construire des widgets personnalisés dans Flutter, vous pouvez étendre le widget de base ou en créer un nouveau à partir de zéro en utilisant la classe StatefulWidget ou la classe StatelessWidget. Vous pouvez définir les

comportements et les styles souhaités dans les méthodes build() et définir les états de votre widget dans la classe d'état associée si nécessaire.

Voici un exemple de widget personnalisé qui affiche un bouton avec un style personnalisé :

```
class CustomButton extends StatelessWidget {
  final String text;
  final Function onPressed;

  CustomButton({this.text, this.onPressed});

  @override
  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(8.0),
      decoration: BoxDecoration(
        color: Colors.lightBlue,
        borderRadius: BorderRadius.all(Radius.circular(5.0)),
      ),
      child: Material(
        color: Colors.transparent,
        child: InkWell(
          onTap: onPressed,
          child: Container(
            padding: EdgeInsets.all(8.0),
            child: Text(
              text,
              style: TextStyle(
                color: Colors.white,
                fontWeight: FontWeight.bold,
                fontSize: 18.0,
              ),
            ),
          ),
        ),
      ),
    );
  }
}
```

Vous pouvez utiliser ce widget personnalisé en le définissant dans votre arbre de widgets, comme ceci :

```
CustomButton(
  text: "Appuyez sur moi",
  onPressed: () {
    // Action à effectuer lorsque le bouton est pressé
  },
)
```

Internationalisation et localisation

L'internationalisation (i18n) est le processus d'adaptation d'une application pour être utilisée dans différents pays et cultures, en prenant en compte les différences linguistiques et culturelles. La localisation (l10n) est le processus de traduction de l'application pour une langue ou une région spécifique.

Dans Flutter, l'internationalisation peut être implémentée en utilisant des packages tels que flutter_localizations, qui offre une prise en charge intégrée pour les traductions, les plannings régionaux et les conventions de format. Les chaînes de traduction peuvent être définies dans des fichiers de ressources externes, puis mappées à des clés dans le code pour une utilisation dynamique dans l'interface utilisateur.
Voici un exemple d'implémentation de l'internationalisation dans Flutter :

1. Ajouter le package flutter_localizations à pubspec.yaml :

```
dependencies:
  flutter_localizations:
    sdk: flutter
```

2. Créer un fichier de traduction, par exemple intl_en.arb :

```
{
  "title": "Hello World!",
  "message": "Welcome to Flutter!"
}
```

3. Ajouter un générateur de traductions dans le fichier pubspec.yaml :

```
flutter:
  generate:
    # Adds a generate section to pubspec.yaml
    - lib/l10n/intl_messages.dart
```

4. Générer des classes Dart à partir des fichiers de traduction :

```
flutter pub pub run intl_translation:generate_from_arb --output-
dir=lib/l10n --no-use-deferred-loading lib/l10n/intl_*.arb
```

5. Utiliser les traductions dans le code Dart :

```dart
import 'package:flutter/widgets.dart';
import 'package:myapp/l10n/intl_messages.dart';

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      localizationsDelegates: [
        IntlMessages.delegate,
      ],
      supportedLocales: [
        Locale('en', ''),
      ],
      home: Scaffold(
        appBar: AppBar(
          title: Text(IntlMessages.of(context).title),
        ),
        body: Center(
          child: Text(IntlMessages.of(context).message),
        ),
      ),
    );
  }
}
```

Cet exemple montre comment utiliser les fichiers de traduction pour définir des traductions de chaînes de caractères, puis comment utiliser ces traductions dans le code en utilisant le code généré pour accéder aux traductions.

Jour 1

1. Introduction à Flutter :
    - Présentation de Flutter :
        - Qu'est-ce que Flutter et pourquoi est-il populaire
        - Fonctionnalités et avantages de Flutter
    - Mise en place de l'environnement de développement :
        - Installation du SDK Flutter
        - Installation d'un IDE (environnement de développement intégré) tel qu'Android Studio ou Visual Studio Code
        - Installation des plugins et packages requis
    - Introduction au langage de programmation Dart :
        - Présentation du langage de programmation Dart
        - Syntaxe de base et types de données
        - Variables, fonctions et structures de contrôle
        - Classes, objets et héritage

Cette section vise à fournir une introduction complète à Flutter et à l'environnement de développement. L'instructeur peut également passer en revue les bases du langage de programmation Dart et sa syntaxe pour fournir une base pour la création d'applications Flutter. Cela préparera le terrain pour le reste du cours et donnera aux étudiants une compréhension claire de ce qu'ils apprendront dans les jours à venir.

Jour 2

1. Création d'interfaces utilisateur avec des widgets Flutter :
    - Comprendre les widgets Flutter et sa hiérarchie :
        - Présentation des widgets Flutter
        - Comprendre l'arborescence des widgets et sa structure
    - Création d'interfaces utilisateur de base à l'aide de widgets Flutter :
        - Utilisation de différents types de widgets (Container, Text, Image, etc.) pour créer des interfaces utilisateur de base

        - Utilisation du widget Scaffold pour créer une mise en page d'application de base
        - Utilisation de ListView pour afficher des données dans une liste déroulante
    - Exploration de différents types de widgets :
        - Conteneur : pour créer une zone rectangulaire pouvant contenir d'autres widgets
        - Texte : Pour afficher du texte à l'écran
        - Image : Pour afficher des images
        - Colonne et ligne : pour organiser les widgets sur une ligne verticale ou horizontale
        - Stack : pour superposer des widgets les uns sur les autres
2. Mise en page et style dans Flutter :
    - Comprendre le concept de mise en page Flutter :
        - Comprendre les contraintes et leur influence sur la mise en page
        - Utilisation de BoxConstraints pour contrôler la taille des widgets
    - Utilisation des widgets BoxConstraints et Expanded :
        - Utilisation de BoxConstraints pour définir des contraintes sur les widgets
        - Utilisation du widget étendu pour remplir l'espace restant dans une ligne ou une colonne
    - Application de styles à l'interface utilisateur à l'aide de thèmes et de feuilles de style :
        - Comprendre le concept de thèmes et de feuilles de style
        - Appliquer des thèmes et des styles à l'ensemble de l'application ou à des widgets individuels
        - Remplacer les styles par défaut par des styles personnalisés

Cette section du cours couvrira l'aspect central de la création d'interfaces utilisateur à l'aide de widgets Flutter. Les étudiants découvriront différents types de widgets et comment les organiser à l'aide de concepts de mise en page tels que les contraintes et les widgets étendus. De plus, les étudiants apprendront les thèmes et les feuilles de style et comment appliquer des styles à leur interface utilisateur. À la fin de cette journée, les étudiants seront capables de créer des interfaces utilisateur de base et de leur appliquer des styles.

Jour 3

1. Entrée utilisateur et navigation :
    - Gestion des entrées utilisateur à l'aide de widgets de formulaire :
        - Comprendre les widgets de formulaire et leur utilisation
        - Utilisation de TextField pour obtenir l'entrée de l'utilisateur
        - Utilisation des widgets Case à cocher, Radio et Commutateur pour sélectionner des options
        - Utiliser DropdownButton pour afficher une liste déroulante
    - Naviguer entre les écrans à l'aide des itinéraires et du widget Navigateur :
        - Comprendre le concept d'itinéraires et de navigation
        - Utilisation du widget Navigateur pour naviguer entre les écrans
        - Utilisation d'itinéraires nommés pour accéder à des écrans spécifiques
        - Passer des données entre les écrans
    - Implémentation de Drawer, BottomNavigationBar et d'autres modèles de navigation :
        - Implémentation de Drawer pour afficher un menu de navigation latéral
        - Implémentation de BottomNavigationBar pour afficher une barre de navigation inférieure
        - Implémentation de TabBar pour afficher les onglets
2. Utilisation des API et des données :
    - Récupération de données à partir d'API à l'aide de packages HTTP :
        - Comprendre le concept des API et des requêtes HTTP
        - Récupérer des données à partir d'API à l'aide du package HTTP
        - Analyser les données JSON renvoyées par les API
    - Analyser les données JSON et les utiliser pour créer des interfaces utilisateur dynamiques :
        - Analyser les données JSON et les utiliser pour créer des interfaces utilisateur dynamiques
        - Affichage des données dans une liste à l'aide des widgets ListView et ListTile
        - Afficher des données dans une grille à l'aide de GridView
    - Stockage et récupération de données à l'aide du stockage local et des préférences partagées :
        - Comprendre le concept de stockage local et de préférences partagées
        - Stocker et récupérer des données à l'aide du package shared_preferences

Cette section du cours couvrira la gestion des entrées utilisateur, la navigation et l'utilisation des API et des données. Les étudiants découvriront les widgets de formulaire et comment obtenir des commentaires de l'utilisateur. Ils apprendront également à naviguer entre les écrans et à transmettre des données entre les écrans. De plus, les étudiants apprendront à récupérer des données à partir d'API, à analyser des données JSON et à stocker et récupérer des données à l'aide du stockage local et des préférences partagées. À la fin de cette journée, les étudiants seront capables de créer des interfaces utilisateur dynamiques qui affichent les données des API.

Jour 4

1. Présentation des widgets Flutter :
    - Que sont les widgets Flutter et pourquoi sont-ils importants
    - Comprendre l'arborescence des widgets et comment cela aide à créer une interface utilisateur
    - Widgets Flutter intégrés couramment utilisés tels que conteneur, ligne, colonne, texte, etc.
2. Dispositions de construction dans Flutter :
    - Comprendre les concepts de mise en page de base tels que le rembourrage, les marges et les proportions
    - Utilisation de différents types de dispositions dans Flutter, telles que les dispositions à enfant unique, les dispositions à plusieurs enfants et les dispositions basées sur la pile
    - Implémentation de modèles d'interface utilisateur communs à l'aide de widgets et de mises en page Flutter
3. Travailler avec l'entrée utilisateur :
    - Gestion des entrées utilisateur telles que les tapotements, les défilements et les glissements
    - Comprendre l'architecture événementielle de Flutter et la manière dont elle gère les entrées utilisateur
    - Utilisation de contrôleurs pour gérer les entrées utilisateur et mettre à jour l'interface utilisateur
4. Naviguer entre les écrans :
    - Comprendre les bases de la navigation dans Flutter
    - Implémentation de la navigation à l'aide de Navigator et Routes
    - Pousser et sauter des écrans et transmettre des données entre les écrans

5. Gestion de l'état :
    - Comprendre la gestion des états dans Flutter et pourquoi c'est important
    - Utilisation de setState pour mettre à jour l'interface utilisateur et gérer l'état
    - Comprendre différentes techniques de gestion d'état, telles que InheritedWidget, ScopedModel, Provider et Bloc

À la fin du jour 4, les étudiants devraient avoir une bonne compréhension de la façon de créer une interface utilisateur dans Flutter, de gérer les entrées des utilisateurs, de naviguer entre les écrans et de gérer l'état. Ils devraient être capables de créer des applications Flutter simples avec plusieurs écrans et entrées utilisateur.

Jour 5

1. Travailler avec les API et la mise en réseau :
    - Comprendre les API et pourquoi elles sont importantes pour les applications mobiles modernes
    - Envoi de requêtes HTTP aux API à l'aide du package Http de Dart
    - Analyser les données JSON des API et les utiliser dans les applications Flutter
2. Données persistantes :
    - Comprendre la persistance des données locales et pourquoi c'est important
    - Utilisation de SQLite et du package sqflite pour conserver les données dans une application Flutter
    - Stockage et récupération de données à l'aide de SharedPreferences
3. Animations et graphiques :
    - Comprendre les animations dans Flutter et pourquoi elles sont importantes
    - Utilisation de AnimationController et AnimatedWidget de Flutter pour créer des animations
    - Comprendre la bibliothèque graphique de Flutter et comment créer des graphiques personnalisés
    - Utilisation de Canvas pour créer des animations et des graphiques personnalisés

4. Déploiement des applications Flutter :
    - Comprendre les différentes options de déploiement pour les applications Flutter
    - Empaquetage et création d'applications Flutter pour Android et iOS
    - Publication d'applications Flutter sur l'App Store et Google Play Store
5. Meilleures pratiques et conseils :
    - Comprendre les meilleures pratiques pour écrire du code propre et maintenable
    - Utiliser des outils de débogage pour trouver et corriger des bogues
    - Trucs et astuces pour améliorer les performances et réduire la taille des applications

À la fin du jour 5, les étudiants devraient avoir une compréhension complète de Flutter et être capables de créer et de déployer des applications Flutter du monde réel. Ils doivent avoir les connaissances et les compétences nécessaires pour poursuivre leur parcours d'apprentissage et créer des applications encore plus avancées.